Efficient Monitoring Algorithms for Various Logical Formalisms in Multiple Monitoring Domains

Thesis Proposal

Patrick O'Neil Meredith

University of Illinois

Abstract. Runtime monitoring is a quickly growing technique for providing many of the guarantees of formal verification, but in a manner that is scalable. Prior work on efficient monitoring focused primarily on finite state properties. Non-finite state techniques existed, but added orders of magnitude of runtime overhead on the monitored system. The vast majority of runtime monitoring has also been limited to the application domain, with violations of safety properties only found on the actual trace of a given program. This proposal describes a plan of research to demonstrate that various logical formalisms, including those more powerful than finite logics, can be efficiently monitored in multiple monitoring domains. The demonstrated monitoring domains run the gamut from the application level with the Java and C/C++ programming languages, to monitoring traces *predicted* from a given run of a program, to hardware based monitors designed to ensure proper peripheral operation. The logical formalisms include multicategory finite state machines, extended regular expressions, past-time linear temporal logic with optimization for hardware based monitors, context-free grammars, linear temporal logic with both past and future operators, and string rewriting. This combination of domains and logical formalisms show that monitoring can be both useful and efficient beyond checking finite-state based properties in a flat execution trace of an application.

1 Problem Description

Runtime monitoring of requirements, most often specified as safety policies, can increase the reliability of the resulting hardware or software systems. There is an increasingly broad interest in uses of monitoring in software development and analysis, as reflected, for example, by abundant approaches proposed recently ([3,10,14,17,26,28,32,35,44,48] among others), and also by the runtime verification (RV) and the formal aspects of testing (FATES) initiatives [8,34,35,37,38,63] among many others. Most techniques, however, have either been focused on finite-state properties or have added orders of magnitude overhead to the monitored systems. Whatever the power of the formalism used, all of the mentioned systems fix the choice of formalism. The proposed research shows that many possible logical formalisms can be used, including those more powerful than finite state, while still maintaining efficiency.

Predictive runtime analysis [21, 24] is able to run a program, collect logs, and reconstruct a causal model of the program that can be used to infer *possible* executions of a programs. These inferred executions can be compared against the formal properties designed for monitoring software applications. The proposed research shows that predictive analysis, which was previously applied only to race detection and atomicity checking, can be applied to formally stated properties written in any of the logical formalisms described here. It also improves the efficiency of predictive runtime analysis by orders of magnitude, making it applicable to real world programs.

Hardware approaches to monitoring have seen less active research. Most attempts have been for the purpose of performance measurement or temperature control rather than the safety properties with which we are concerned. [47] is an approach that generates monitors from formal properties that are implemented in hardware, but these hardware monitors are actually used to monitor software programs, making it fit more with the software approaches mentioned above than the hardware based monitoring that is the focus of this proposed research.

2 Background

To understand the current results presented in Section 4 and research plan outlined in Section 5 some background on monitoring, parametric monitoring, and predictive analysis must be explained. The background on parametric monitoring is focused through the lens of JavaMOP, upon which this work is based. The background on monitoring presented here should be sufficient to understand the hardware peripheral monitoring of the BusMOP system presented in Section 4.2.

2.1 Monitoring

```
UnsafeMapIterator(Map m, Collection c, Iterator i){
  event create_coll after(Map m) returning(Collection c)} :
       (call(* Map.values()) || call(* Map.keySet())) && target(m) {}
  event create_iter after(Collection c) returning(Iterator i) :
        call(* Collection+.iterator()) && target(c) {}
  event use_iter before(Iterator i) : call(* Iterator+.next()) && target(i) {}
  event update_map after(Map m) : call(* Map.remove*(..)) || call(* Map.put*(..))
      || call(* Map.putAll*(..)) || call(* Map.clear())) && target(m) {}
  fsm: start [ create_coll -> s1 ]
       s1    [ update_map -> s1, create_iter -> s2 ]
       s2    [ use_iter -> s2, update_map -> s3    ]
       s3    [ update_map -> s3, uset_iter -> end  ]
       end   []
    @end { System.out.println("fsm:  Accessed invalid Iterator!"); __RESET; }
  ere: create_coll update_map* create_iter use_iter* update_map+ use_iter
    @match { System.out.println("ere: Accessed Invalid Iterator!"); __RESET; }
  cfg: S -> create_coll Updates create_iter Nexts update_map Updates use_iter,
       Nexts -> Nexts use_iter | epsilon
       Updates -> Updates update_map | epsilon
    @match { System.out.println("cfg: Accessed Invalid Iterator!"); __RESET; }
  ltl: <>(create_coll and <> (create_iter and <> (update_map and <> use_iter)))
    @validation { System.out.println("ltl: Accessed Invalid Iterator!"); __RESET; }
  ptltl: use_iter ->
         ((<*> (create_iter and (<*> create_coll))) => ((not update_map) S create_iter))
    @violation { System.out.println("ptltl: Accessed Invalid Iterator!"); __RESET; }
}
```

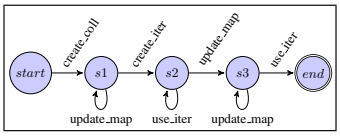


Figure 1. FSM, ERE, CFG, FTLTL, and PTLTL UnsafeMapIterator. Inset: graphical depiction of the property.

Monitoring executions of a system against expected properties plays an important role not only in different stages of software development, e.g., testing and debugging, but also in the deployed system as a mechanism to increase system reliability. This is achieved by allowing the monitors to perform *recovery* actions in the case that a specification is matched, or fails to match. Numerous approaches, such as [3, 6, 9, 19, 20, 28, 32, 36, 48], have been proposed to build effective and efficient monitoring solutions for different applications. More recently, monitoring of parametric specifications, i.e., specifications with free variables, has received increasing interest due to its effectiveness at capturing system behaviors, such as the one presented in Figure 1, which encapsulates the proper use of Map Iterators.

It is highly non-trivial to monitor such parametric specifications efficiently. It is possible to see a tremendous number of parameter instances during the execution of a monitored program. For example, it is not uncommon to see hundreds of thousands of iterators in a program, which will generate hundreds of thousands of parameter instances in the UnsafeMapIterator specification in Figure 1.

Several approaches have been introduced to support the monitoring of parametric specifications, including Eagle [9], Tracematches [3, 6], PQL [48], and PTQL [32]. However, they are all limited in terms of supported specification formalisms. Other techniques, e.g., Eagle, Tracematches, PQL and PTQL, follow a formalism-dependent approach, that is, they have their parametric specification formalisms hardwired, e.g., regular patterns (like Tracematches), context-free patterns (like PQL) with parameters, etc., and then develop algorithms to generate monitoring code for the particular formalisms. Although this approach provides a feasible solution to monitoring parametric specifications, we argue that it not only has limited expressiveness, but also causes unnecessary complexity in developing optimal monitor generation algorithms, often leading to inefficient monitoring. In fact, the experiments summarized in [18, 42, 49, 51] show that JavaMOP generates more efficient monitoring code than other existing tools; much of this efficiency is due to two main optimizations presented in Section 4.

Figure 1 shows a JavaMOP specification of the UnsafeMapIterator property. The idea of UnsafeMapIterator is to catch an intricate safety property of Java. There are several methods to create Collection (essentially sets) from Java

Maps. One may then create Java Iterators to traverse these Collections. However, if the Map is updated, the Iterators are invalidated.

The specification uses five different formalisms: finite state machines (FSM), extended regular expressions (ERE), context-free grammars (CFG), future-time linear temporal logic (FTLTL), and past-time linear temporal logic (PTLTL). Because each of the properties in Figure 1 is the same, five messages will be reported whenever an Iterator is incorrectly used after an update to the underlying Map. We show all five of them to emphasize the formalism-independence of our approach. Under normal circumstances a user would chose just one formalism.

On the first line, we name the specified property and give the parameters used in the specification. Then we define the involved events using the AspectJ syntax. For example, create_coll is defined as the return value of functions values and keyset of Map. We adopt AspectJ syntax to define events in JavaMOP because it is an expressive language for defining observation points in a Java program. As mentioned, every event may instantiate some parameters at runtime. This can be seen in Figure 1: create_coll will instantiate parameters m and c using the target and the return value of the method call. When one defines a pattern or formula there are implicit events, which must begin traces; we call them *monitor creation* events. For example, in a pattern language like ERE, the monitor creation events are the first events that appear in the pattern. We assume a semantics where events that occur before monitor creation events are ignored.

2.2 Parametric Slicing

#	Event	#	Event
1	create_coll$\langle m_1, c_1\rangle$	7	update_map$\langle m_1\rangle$
2	create_coll$\langle m_1, c_2\rangle$	8	use_iter$\langle i_2\rangle$
3	create_iter$\langle c_1, i_1\rangle$	9	create_coll$\langle m_2, c_3\rangle$
4	create_iter$\langle c_1, i_2\rangle$	10	create_iter$\langle c_3, i_4\rangle$
5	use_iter$\langle i_1\rangle$	11	use_iter$\langle i_4\rangle$
6	create_iter$\langle c_2, i_3\rangle$		

Figure 2. Possible execution trace over the events specified in UnsafeMapIterator.

JavaMOP automatically synthesizes AspectJ instrumentation code from the specification, which is weaved into the program we wish to monitor by any standard AspectJ compiler. In this way, executions of the monitored program will produce traces made up of events defined in the specification, as those in Figure 1. Consider the example eleven event trace in Figure 2 over the events defined in Figure 1. The # column gives the numbering of the events for easy reference. Every event in the trace starts with the name of the event, e.g., create_coll, followed by the parameter binding information, e.g., $\langle m_1, c_1\rangle$ that binds parameters m and c with a map object m_1 and a collection c_1, respectively. Such a trace is called a *parametric trace* since it contains events with parameters.

Our approach to monitoring parametric traces against parametric properties is based on the observation that each parametric trace actually contains multiple *non-parametric trace slices*, each for a particular parameter binding instance. Intuitively, a slice of a parametric trace for a particular parameter binding consists of names of all the events that have identical or *less informative* parameter bindings. Informally, a parameter binding b_1 is identical or less informative than a parameter binding b_2 if and only if the parameters for which they have bindings agree, and b_2 binds either an equal number of parameters or more parameters: parameter $\langle m_1, c_2\rangle$ is less informative than $\langle m_1, c_2, i_3\rangle$ because the parameters they both bind, m and c, agree on their values, m_1 and c_2, respectively, and $\langle m_1, c_2, i_3\rangle$ binds one more parameter. From here on we will simply say less informative to mean identical or less informative. Figure 3 shows the trace slices and their corresponding parameter bindings contained in the trace in Figure 2. The Status column denotes the monitor output category that the slice falls into (for ERE). In this case everything but the slice for $\langle m_1, c_1, i_2\rangle$, which matches the property, is in the "?" (undecided) category. For example, the trace for the binding $\langle m_1, c_1\rangle$ contains create_coll update_map (the first and seventh events in the trace) and the trace for the binding $\langle m_1, c_1, i_2\rangle$ is create_coll create_iter update_map use_iter (the first, fourth, seventh, and eighth events in the trace).

Based on this observation, our approach creates a set of monitor instances during the monitoring process, each handling a trace slice for a parameter binding. Figure 4 shows the set of monitor instances created for the trace in Figure 2, each monitor labeled by the corresponding parameter binding. This way, the monitor *does not need to handle the parameter information* and can employ any existing technique for ordinary, non-parametric traces, including state

Instance	Slice	Status
$\langle m_1 \rangle$	update_map	?
$\langle m_1, c_1 \rangle$	create_coll update_map	?
$\langle m_1, c_2 \rangle$	create_coll update_map	?
$\langle m_2, c_3 \rangle$	create_coll	?
$\langle m_1, c_1, i_1 \rangle$	create_coll create_iter use_iter update_map	?
$\langle m_1, c_1, i_2 \rangle$	create_coll create_iter update_map use_iter	match
$\langle m_1, c_2, i_3 \rangle$	create_coll create_iter update_map	?
$\langle m_2, c_3, i_4 \rangle$	create_coll create_iter use_iter	?

Figure 3. Slices for the trace in Figure 2.

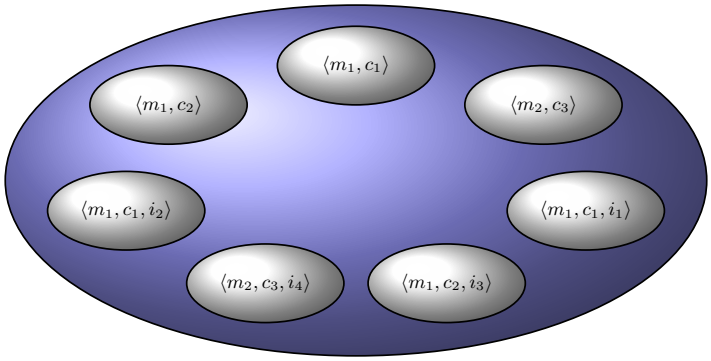


Figure 4. A parametric monitor with corresponding parameter instance monitors.

machines and push-down automata, providing a formalism-independent way to check parametric properties. When an event comes, our algorithm will dispatch it to related monitors, which will update their states accordingly. For example, the seventh event in Figure 2, update_map$\langle m_1 \rangle$, will be dispatched to monitors for $\langle m_1, c_1 \rangle$, $\langle m_1, c_2 \rangle$, $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$. New monitor instances will be created if the event contains new parameter instances. For example, when the third event in Figure 2, create_iter$\langle c_1, i_1 \rangle$, is received, a new monitor will be created for $\langle m_1, c_1, i_1 \rangle$ by combining $\langle m_1, c_1 \rangle$ in the first event with $\langle c_1, i_1 \rangle$.

An algorithm to build parameter instances from observed events, like the one introduced in [22], may create many useless monitor instances leading to prohibitive runtime overheads. For example, Figure 3 does not need to contain the binding $\langle m_1, c_3, i_4 \rangle$ even though it can be created by combining the parameter instances of update_map$\langle m_1 \rangle$ (the seventh event) and create_iter$\langle c_3, i_4 \rangle$ (the tenth event). It is safe to ignore this binding here because m_1 is not the underlying map for c_3, i_4. It is critical to minimize the number of monitor instances created during monitoring. The advantage is twofold: (1) that it reduces the needed memory space, and (2), more importantly, monitoring efficiency is improved since fewer monitors are triggered for each received event. JavaMOP uses two main algorithms in order to prevent the creation of instances that are known to be unneeded and to remove those that become unneeded during execution; these are presented in Section 4.1.

2.3 Predictive Analysis

Concurrent systems in general and multithreaded systems in particular may exhibit different behaviors when executed at different times. This inherent nondeterminism makes multithreaded programs difficult to analyze, test and debug. Several approaches to finding concurrency errors have been proposed, such as [30, 54, 59, 60, 62, 66, 67]. Most of these approaches focus on atomicity or data races rather than generic safety properties. [62] does allow for general purpose properties expresses using ast time linear temporal logic, but the properties are not parametric, and the implementation was very prototypical and based on a weaker causal relation. Predictive analysis is able to detect, correctly, concurrency errors from observing execution traces of multithreaded programs. By "correct" or "sound" prediction of errors we mean that there are *no false alarms*. The program is automatically instrumented to emit runtime events which are used

to predict violations in unobserved runs. The particular execution that is observed need *not* hit the error; yet, errors in other executions can be correctly predicted together with counter-examples leading to them.

Our predictive runtime analysis technique, implemented in the tool RV-Predict, can be understood as a hybrid of testing and model checking. Testing because one runs the system and observes its runtime behavior in order to detect errors, and model checking because the special causality with lock-atomicity extracted from the running program can be regarded as an abstract model of the program, which can further be investigated exhaustively by the observer in order to detect potential errors. A more in depth presentation of how RV-Predict build causal models and predicts errors is presented in Section 4.3.

3 Research Goals

The goal of this proposed research is to facilitate the growth of runtime verification techniques in the software design cycle, as well as to make implemented systems more robust. The work on the efficiency of JavaMOP and RV-Predict greatly aids in the early development cycle, adding in both testing and debugging new designs. Increasing the efficiency of JavaMOP also enables JavaMOP to be used in deployed systems, ensuring safe and secure operation at runtime through the use of recovery handlers. BusMOP ensures that hardware peripherals cannot compromise the health of a running system, and is in this respect similar to JavaMOP in deployed systems. Developing new algorithms for more powerful logical formalisms allows users more freedom in expressing ever more complex specifications, further increasing the utility of runtime verification techniques.

4 Completed Research

Currently, two optimizations to JavaMOP have been implemented that result in a very efficient system. BusMOP and RV-Predict show that monitoring is useful in domains other than the direct monitoring of application traces, while the logics outlined in this section show that monitoring multiple logical formalisms, including those more powerful than finite-state can be monitored.

Algorithm $\mathcal{EN}_{fsm}(FSM = (\mathcal{E}, S, s_0, \delta))$
Globals: mapping $\mathcal{V}_\mu : S \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$
mapping $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$
Initialization: $\mathcal{V}_\mu(s) \leftarrow \emptyset$ for any $s \in s$
$\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) \leftarrow \emptyset$ for any $e \in \mathcal{E}$
function main()
1 compute_enables$(s_0, \emptyset)$
function compute_enables(s, μ)
1 foreach defined $\delta(s, e)$ do
2 : $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) \leftarrow \mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) \cup \{\mu\}$
3 : let $\mu' \leftarrow \mu \cup \{e\}$
4 : if $\mu' \notin \mathcal{V}_\mu(s)$
5 : : $\mathcal{V}_\mu(s) \leftarrow \mathcal{V}_\mu(s) \cup \{\mu'\}$
6 : : compute_enables$(\delta(s, e), \mu')$
7 : endif
8 endfor

Figure 5. FSM $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ Computation Algorithm [18].

4.1 JavaMOP

In order to be as efficient as possible, JavaMOP makes use of two pieces of information generated by the logic plugins described below in order to reduce the number of monitor instances alive at any one time during monitoring. These pieces of information are the enable set [18], and its dual, the coenable set [42].

$$
\begin{aligned}
\mathrm{G}(\epsilon) &= \{\emptyset\} \\
\mathrm{G}(t) &= \{\{t\}\} \\
\mathrm{G}(A) &= \textstyle\bigcup_{A \to \beta} \mathrm{G}(\beta) \\
\mathrm{G}(\beta_1\beta_2) &= \{S \cup T \mid S \in \mathrm{G}(\beta_1), T \in \mathrm{G}(\beta_2)\} \\
\mathrm{P}(\gamma) &= \{S \cup T \mid A \to \beta_1\gamma\beta_2, S \in \mathrm{P}(A), T \in \mathrm{G}(\beta_1)\} \\
\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) &= \mathrm{P}(e)
\end{aligned}
$$

Figure 6. CFG $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ Defining Equations

The enable set essentially keeps track of which parameters must be seen before a given event can occur if a given result category, such as match, can still be reached. The idea is that if the categories we are interested in cannot be reached by a given monitor instance, then there is no reason to create that instance. The coenable set, then, produces the dual effect. It is able to determine which monitor instances that have already been created may be removed because they can no longer reach a category of interest. The coenable set encodes which parameter objects must still exist in the system for a category to be reached after a given event. If a key parameter object no longer exists (because it has been garbage collected) it is safe to remove the entire monitor instance. Figures 5 and 6 show an algorithm to compute the enable sets for finite-state machines and mathematical equals to compute the enable set for context-free grammars, respectively. These algorithms can be used to compute coenable sets simply by reversing the finite-state machine or mirroring the context-free grammar productions. Figure 7 shows performance results for JavaMOP with only the enable set algorithm (E) and with the addition of the coenable set based monitor instance garbage collection algorithm (CE) in comparison with tracematches (TM) [6], the most efficient monitoring system other than JavaMOP of which we are aware. All benchmarks are from the DaCapo benchmark suite [13].

4.2 BusMOP

Every monitoring domain has its own specific issues: BusMOP must deal with interfacing with buses at a hardware level. There is also complexity in the mechanisms for recovery actions, which require specialized hardware modules. A complete introduction to the PCI bus used in the current implementation of BusMOP can be found in both [51] and [56]. Here we discuss only the design of the BusMOP monitoring device. Particular care must be taken to support certain features of the current MOP logical formalism semantics that expect serialized events. To use BusMOP, one writes a property for a specific peripheral, say p, that is plugged into the bus. The property is then synthesized onto an FPGA that is also plugged into the bus. A small program is used to write the proper value of the Base Access Register (BAR) for p to the FPGA (see [56] or [51]for more explanation on the BARs).

Monitoring Device The current version of BusMOP is designed for the Xilinx ML455 board [68], but adapting the generated code to different FPGA boards takes minimal effort. The monitoring device uses a mixed VHDL/Verilog register transfer level (RTL) description. The board is outfitted with a Virtex-4 FPGA and it can be plugged into a standard 3.3V PCI/PCI-X socket. The FPGA implements both a slave and a master peripheral module, together with the monitoring modules. Events for the system are specified in terms of read/write data transfers on the bus and interrupt requests; the device continuously "sniffs" all ongoing activities on the bus, and is therefore able to monitor communication for all other peripherals located on the same bus segment. Whenever a failure to meet the specification is detected, the device can execute a recovery action using strategies based on the detected error.

For a vast category of errors that involve incorrect interaction between the peripheral and its software driver, it is often possible to recover from the failure by forcing the peripheral into a consistent state. The monitoring device implements a master module, and can therefore initiate transactions on the bus. For example, consider a common type of

		HASNEXT			UNSAFEITER			UNSAFEMAPITER			UNSAFESYNCCOLL			UNSAFESYNCMAP			ALL
(A)	ORIG (sec)	TM	E	CE	TM	E	CE	TM	E	CE	TM	E	CE	TM	E	CE	CE
bloat	3.6	2119	448	116	19194	569	251	∞	1203	178	1359	746	212	1942	716	130	982
jython	8.9	13	0	0	11	0	1	150	18	3	11	1	1	10	0	0	4
avrora	13.6	45	54	55	637	311	118	∞	113	42	75	144	80	54	74	16	275
batik	3.5	3	2	3	355	9	8	∞	8	5	208	9	9	5	3	0	28
eclipse	79.0	-2	4	-1	0	-1	-1	5	-3	0	-4	2	1	∞	-1	-1	0
fop	2.0	200	49	48	350	21	13	∞	58	14	∞	78	25	∞	71	19	133
h2	18.7	89	17	13	128	9	4	1350	21	6	868	21	4	83	20	5	23
luindex	2.9	0	0	1	0	0	1	1	4	1	1	1	1	2	0	0	1
lusearch	25.3	-1	1	0	1	2	2	2	2	0	4	0	1	3	1	1	3
pmd	8.3	176	84	59	1423	162	123	∞	571	188	1818	192	76	∞	144	26	620
sunflow	32.7	47	5	3	7	2	0	9	4	1	13	6	5	17	6	6	6
tomcat	13.8	8	1	1	37	1	1	3	1	1	2	0	1	2	1	3	1
tradebeans	45.5	0	-1	1	1	1	2	5	3	-1	-1	1	2	3	1	5	2
tradesoap	94.4	1	3	0	2	1	1	2	0	1	0	0	1	2	2	5	1
xalan	20.3	4	2	2	27	7	2	10	5	2	3	2	3	4	4	3	4
(B)	ORIG (MB)	TM	E	CE	TM	E	CE	TM	E	CE	TM	E	CE	TM	E	CE	CE
bloat	4.9	56.8	19.3	13.2	7.7	146.8	79.0	∞	173.4	56.1	6.8	127.9	48.3	6.9	55.4	12.7	340.9
jython	5.3	5.7	4.6	4.8	4.9	4.6	4.8	6.0	19.5	4.7	5.3	4.5	4.4	5.9	4.8	5.1	4.7
avrora	4.7	4.6	12.4	9.1	4.4	136.2	15.8	∞	14.7	8.5	4.3	28.0	12.6	4.4	13.0	4.9	22.3
batik	79.1	79.2	78.7	79.3	75.2	93.6	86.6	∞	91.2	79.6	78.2	93.2	85.1	79.9	86.9	76.7	104.3
eclipse	95.9	100.8	107.6	97.1	98.3	100.0	110.3	106.9	93.8	101.1	100.4	109.2	90.1	∞	98.6	98.7	98.9
fop	20.7	97.4	47.1	52.5	24.3	24.2	29.4	∞	69.2	28.1	∞	54.8	24.8	∞	55.9	25.2	47.5
h2	265.0	267.8	598.5	565.2	267.2	266.2	262.4	312.4	688.3	268.2	271.4	690.3	265.5	271.0	718.3	270.0	283.7
luindex	6.8	5.6	5.5	5.6	6.3	6.9	6.8	7.4	8.2	6.9	7.4	7.4	7.5	7.1	7.4	11.0	11.8
lusearch	4.6	4.7	4.4	4.8	4.6	4.8	4.2	4.0	4.3	4.8	4.5	4.5	4.6	4.6	4.8	4.7	4.7
pmd	18.0	56.9	59.8	48.5	17.2	146.3	86.4	∞	212.7	93.6	20.3	238.4	84.6	∞	117.1	32.9	420.0
sunflow	4.4	4.5	4.8	4.9	4.8	4.3	4.7	4.7	4.4	4.4	5.1	4.3	4.9	4.5	4.7	4.5	4.6
tomcat	11.6	11.4	12.3	11.4	12.5	11.0	11.5	11.9	11.4	11.0	11.3	11.3	11.3	11.4	11.4	11.8	11.8
tradebeans	63.2	62.9	62.7	62.1	63.7	63.9	64.1	63.3	62.5	62.7	63.2	62.8	62.0	64.0	62.8	64.0	62.5
tradesoap	64.1	61.8	62.3	63.3	63.4	63.1	64.4	64.1	63.5	62.0	60.7	65.0	65.9	65.5	64.5	65.6	64.5
xalan	4.9	4.9	5.0	5.1	4.9	4.9	4.9	4.9	4.5	4.9	5.0	4.8	5.0	5.1	4.9	4.9	5.0

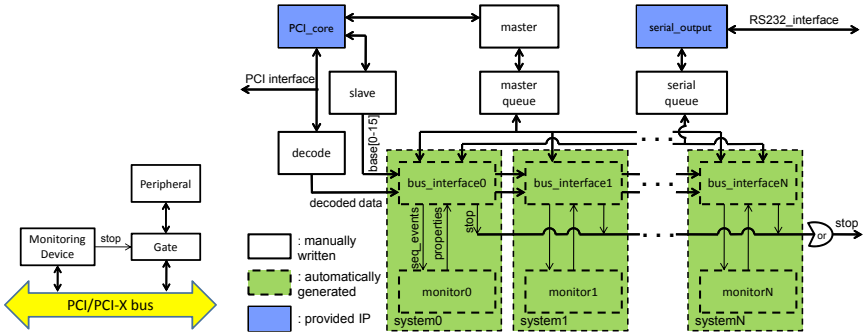
Figure 7. Comparison of Tracematches (TM), JavaMOP with enable set optimization (E), and JavaMOP with both enable and coenable optimizations: (A) average *percent* runtime overhead; (B) total peak memory usage in MB. (convergence within 3%, ∞: not terminated after 1 hour)

error, where the driver fails to validate some input from the user and as a result writes an invalid value to a register in the peripheral. We can recover by rewriting the register with a valid value. However, if the error is caused by a fault in the peripheral hardware, interacting with registers may not be enough to bring the peripheral to a consistent and safe state.

To handle peripherals that cannot be put into a consistent and safe state, a hardware device, the *peripheral gate* [55], is used that is able to force the REQ# signal from the peripheral to the bus arbiter to be high. Hence, the peripheral never receives the grant and it is prohibited from initiating any further transaction on the bus.[1] The peripheral gate is implemented based on a PCI extender card, i.e., a debug card that is interposed between the peripheral card and the bus and provides easy access to all signals. A clarifying picture for monitoring of a single peripheral is provided in Fig. 8(a). The monitoring device can output a *stop* signal, which closes the gate when active high; this can be achieved in a specification by setting a variable called stop_reg, described in [51], to '1'. Finally, sometimes the monitoring device cannot perform a suitable recovery action by itself, but there is a higher level actor, such as the OS or the system user, that can provide better recovery; examples include complex software operations such as restarting the driver or the whole PCI stack, and physically interacting with the peripheral. In this case, the best strategy is to communicate the failure to the chosen actor. Additionally, we implemented an RS-232 controller that can be used to send information to the user over a serial connection.

The nature of our implementation is such that if a trace is seen, which does not conform to a specification, as a consequence of a bus transaction, that specific bus transaction cannot be prevented from propagating to the rest of the system. For example, if a faulty peripheral performs a write transaction to an area in main memory which is not supposed to modify, we can detect the error, disconnect the peripheral, and report the failure to the OS/user. However,

[1] While technically it is always possible for a faulty peripheral to disrupt the bus by altering the state of the signals, in practice the described approach is effective since access to the bus is mediated by three-state buffers enabled by GNT#.



(a) Gated Monitoring Device.

(b) Block Diagram.

Figure 8. Monitoring Device.

the information in the overwritten area will be lost. BusMOP has also been used within the context of a System on a Chip (SoC) design platform where it is able to perform actual preventive actions (e.g., disallowing writes) [57].

A simplified block diagram for the monitoring device is shown in Fig. 8(b). An explanation of these blocks can be found in [51] or [56]. Note that, in general, BusMOP places 0% runtime overhead on the system it monitors.

4.3 Predictive Analysis

The Predictive Analysis implemented in RV-Predict is able to scalably predict data races. It is also able to predict general purpose properties, but currently they must be written as Java classes by hand. While this proves that the technique is feasible, the goal is to improve upon the user interface for general purpose properties as outlined in Section 5. As follows is a brief description of how RV-Predict successfully predicts errors in a scalable manner. Of particular importance are the necessitates of minimizing the number of trace reversals, as well as handling traces that are much too large to fit in main memory, features not properly handled in [24], which resulted in its not being able to handle real world programs. The work presented here is covered in [52].

Causal Slicing We briefly describe our technique for extracting from an execution trace of a multithreaded system the sliced causality relation corresponding to some property of interest φ. Our technique is *offline*, in the sense that it takes as input an already generated execution trace; that is because causal slicing must traverse the trace backwards. Our technique consists of two steps: ***(1)*** all the irrelevant events (those which are neither property events nor events on which property events are dependant) are removed from the original trace, obtaining the *(φ)-sliced trace*; and ***(2)*** a *vector clock (VC)* based algorithm is applied on the sliced trace to capture the sliced causality partial order.

–Extracting Slices– Our goal here is to take a trace ξ and a property φ, and to generate a trace ξ_φ obtained from ξ filtering out all its events which are irrelevant for φ. When slicing the execution trace, one must nevertheless keep all the property events. Moreover, one must also keep any event e with $e \left(\sqsubset_{ctrl} \cup \sqsubset_{data}\right)^+ e'$ for some property event e'. This can be easily achieved by traversing the original trace backwards, starting with ξ_φ empty and accumulating in ξ_φ events that either are property events or have events depending on them already in ξ_φ. One can employ any off-the-shelf analysis tool for data- and control- dependence; e.g., RV-Predict uses termination-sensitive control dependence [23].

To understand the process, consider the example in Fig. 9, threads T_1 and T_2 are executed as shown by the solid arrows (A), yielding the event sequence "e_1, e_2, e_3, e_4, e_5, e_6" (B). Suppose the property to check refers only to y; the property events are then e_1, e_5, and e_6. Events e_2 and e_3 are immediately marked as relevant, since $e_2 \sqsubset_{data} e_3 \sqsubset_{ctrl} e_5$. If only closure under control- and data-dependence were used to compute the relevant events, then e_4 would appear to be irrelevant, so one may conclude that "e_2, e_6, e_1, e_3, e_5" is a sound permutation; there is, obviously, no execution that can produce that trace, so one reported a false alarm if that trace violated the original property on y. Consequently, e_4 is also a relevant event and $e_3 \sqsubset_{rlvn} e_4$.

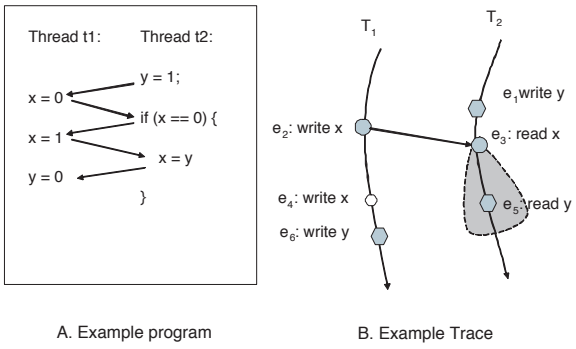


Figure 9. Example for relevance dependence

Unfortunately, one backwards traversal of the trace does not suffice to correctly calculate all the relevant events. Reconsider Fig. 9. When the backward traversal first reaches e_4, it is unclear whether e_4 is relevant or not, because we have not seen e_3 and e_2 yet. Thus a second scan of the trace is needed to include e_4. Once e_4 is included in ξ_φ, it may induce other relevance dependencies, requiring more traversals of the trace to include them. This process would cease only when no new relevant events are detected and thus resulting sliced trace stabilizes. If one misses relevant events like e_4 then one may "slice the trace too much" and, consequently, one may produce false alarms. Because at each trace traversal some event is added to ξ_φ, the worse-case complexity of the sound trace slicing procedure is square in the number of events. Since execution traces can be huge, in the order of billions of events[2], any trace slicing algorithms that is worse than linear may easily become prohibitive. For that reason, RV-Predict traverses the trace only once during slicing, thus achieving an approximation of the complete slice that can, in theory, lead to false alarms. However, our experiments show that this approximation is actually very precise in practice: we have yet to find a false alarm in any of our experiments.

–*Vector Clocking*– Vector clocks [45] are routinely used to capture causal partial orders in distributed and concurrent systems. A *VC*-based algorithm was presented in [62] to encode a conventional multithreaded-system "happen-before" causal partial order on the unsliced trace. We next adapt that algorithm to work on our sliced trace and thus to capture the sliced causality. Recall that a vector clock (*VC*) is a function from threads to integers, $VC : T \to Int$. We say that $VC \leq VC'$ iff $\forall t \in T, VC(t) \leq VC'(t)$. The $\max$ function on *VCs* is defined as: $\max(VC_1, ..., VC_n)(t) = \max(VC_1(t), ..., VC_n(t))$ ([62]).

Before we explain our *VC* algorithm, let us introduce our event and trace notation. An *event* is a mapping of *attributes* into corresponding *values*. One event can be, e.g., $e_1 : (counter = 8, thread = t_1,\ stmt = L_{11},\ type = write,\ target = a,\ state = 1)$, which is a write on location a with value 1, produced at statement L_{11} by thread t_1. One can include more information into an event by adding new attribute-value pairs. We use $key(e)$ to refer to the value of attribute *key* of event e. To distinguish different occurrences of events with the same attribute values, we add a designated attribute to every event, *counter*, collecting the number of previous events with the same attribute-value pairs (other than the *counter*). The trace for the vector clocking step is the φ-sliced trace ξ_φ obtained in Section 4.3.

Intuitively, vector clocks are used to track and transmit the causal partial ordering information in a concurrent computation, and are typically associated with elements participating in such computations, such as threads, processes, shared variables, messages, signals, etc. If *VC* and *VC'* are vector clocks such that $VC(t) \leq VC'(t)$ for some thread t, then we can say that *VC'* has newer information about t than *VC*. In our *VC* technique, every thread t keeps a vector clock, VC_t, maintaining information about all the threads obtained both locally and from thread communications (reads/writes of shared variables). Every shared variable is associated with two vector clocks, one for writes (VC_x^w) used to enforce the order among writes of x, and one for all accesses (VC_x^a) used to accumulate information about all accesses of x. They are then used together to keep the order between writes and reads of x. Every property event e

[2] RV-Predict compresses traces to keep sizes manageable. Reversing the trace is done at logging time by outputting a buffer of events backwards into separate archives. The archives are then read by the trace slicer in reverse order.

found in the analysis is associated a *VC* attribute, which represents the computed causal partial order. We next show how to update these VCs when an event e is encountered during the analysis (the third case can overlap the first two; if so, the third case will be handled first):

1. *type*$(e) =$ *write*, *target*$(e) = x$, *thread*$(e) = t$ (the variable x is written in thread t) and x is a shared variable. In this case, the write vector clock VC_x^w is updated to reflect the newly obtained information; since a write is also an access, the access *VC* of x is also updated; we also want to capture that t committed a causally irreversible action, by updating its *VC* as well: $VC_t \leftarrow VC_x^a \leftarrow VC_x^w \leftarrow \max(VC_x^a, VC_t)$.
2. *type*$(e) =$ *read*, *target*$(e) = x$, $thread(e) = t$ (the variable x is read in t), and x is a shared variable. Then the thread updates its information with the write information of x (we do not want to causally order reads of shared variables!), and x updates its access information with that of the thread: $VC_t \leftarrow \max(VC_x^w, VC_t)$ and then $VC_x^a \leftarrow \max(VC_a^x, VC_t)$.
3. e is a property event and $thread(e) = t$. In this case, let $VC(e) := VC_t$. Then $VC_t(t)$ is increased to capture the intra-thread total ordering: $VC_t(t) \leftarrow VC_t(t) + 1$.

Race and Generic Property Detection The basic idea of race detection is simple: check for accesses to the same variable with incomparable *VCs*. However, it is easy to note that this has quadratic worst case complexity, because each access must be compared against every other access. Clearly, when billions of accesses may occur in a trace, this is unacceptable. Not only would this be unbearable slow, but it would be impossible to even fit the accesses in memory to perform the comparisons, as was the case with jPredictor.

To alleviate this, as well as to make it more easy to deal with streaming to and from the disk when memory is overfull, we use the idea of a window of comparisons, ignoring pairs of events that trivially cannot have incomparable vector clocks. If at some point we note the second access, $a_2^{T_1}$ in thread T_1 must occur after the fifth access, $a_5^{T_2}$, in thread T_2 we know that we do not need to check the $a_2^{T_1}$ against any further accesses in thread T_2 because all accesses in a given thread must be totally ordered (and the traces are backwards).

To implement this we use a *set*[3] of search states. Each search state abstracts the notion of checking accesses in two threads. Each search state keeps an iterator to the list of accesses representing one of its two given threads. The algorithm begins by keeping search states for each pair of threads in a set (actually not all threads are known immediately, but we will elide this detail for ease of understanding). Each state is advanced by considering the accesses pointed to by each of its iterators. If the iterators are incomparable, three new search states are added to the set. One state where one iterator is advanced, one where the other iterator is advanced, and one where both iterators are advanced. If the two accesses are incomparable and are not protected by a shared lock, a race is reported. If, one the other hand, the vector clocks of the two accesses in question are ordered, only one of the iterators is advanced, for example, if the access in thread t of the search state must take place before the access in thread t', the iterator pointing to the access from thread t is advanced, and no other states are generated.

This idea is easily extrapolated to generic property detection. One caveat, however, is that the iterators of the search states point to streams of JavaMOP monitoring events rather than accesses to shared variables. Also, rather than keeping iterators to only two threads in a search state, each search state keeps an array of iterators, one to each thread in the program as they are discovered in the trace. Each search state, additionally, keeps a reference to a monitor provided by JavaMOP[4]. When a search state is advanced, a new set of states is created and added to the overall set of states the same as for race detection, save that the advanced iterators are the subset of iterators with incomparable *VCs*, and that states that end up with the same monitor state are collapsed immediately into one chosen representative search state. For each search state thus generated, the event uncovered by advancing one of the iterators is given to its monitor to check for property violation or validation. While this is exponential in the worst case, in practice most search states are collapsed because they have identical monitor states and positions in the event stream. Additionally, the number of events relative to a general purpose property tends to be fair smaller than the number necessary to predict races in programs.

Prediction Results Fig. 10 summarizes the differences in real time and disk usage between the original jPredictor system first presented in [24] and RV-Predict for race prediction as measured on a system with two quad core Xeon

[3] We must use a set to avoid duplicate search states, or the algorithm can quickly explode.

[4] Currently, as mentioned, the monitor must be hand modified to work with RV-Predict, this will be ameliorated in the proposed work (Section 5).

		jPredictor		RV-Predict	
Name	Input	Real Time	Disk Usage	Real Time	Disk Usage
account	-	0:02.07	236K	0:04.31	360K
elevator	-	5:55.29	63M	1:20.31	864K
tsp	map4 2	5:30.87	16M	1:33.44	744K
tsp	map5 2	10:10.19	17M	2:20.95	868K
tsp	map10 2	8:25:04.00	442M	29:27.13	2.8M
huge	-	crash	crash	0:42.22	13M
medium	-	crash	crash	0:06.12	840K
small	-	crash	crash	0:05.99	292K
mixedlockshuge	-	8:13:40.00	250M	0:13.95	2.9M
mixedlocksbig	-	5:44.89	25M	0:07.03	496K
mixedlocksmedium	-	0:08.92	2.7M	0:07.25	308K
mixedlockssmall	-	0:05.46	1.5M	0:05.67	296K

Figure 10. jPredictor Vs. RV-Predict (Race Detection Only)

E5430 processors running at 2.66GHz and 16 GB of 667 MHz DDR2 memory running Redhat Linux. On very small examples jPredictor occasionally outperforms RV-Predict, but on anything substantial RV-Predict is a vast improvement. Account, elevator, and tsp are actual programs used to benchmark parallel systems. Huge, medium, small, and the mixed locks examples are microbenchmarks that we designed to test particularly difficult aspects of race detection, such as millions of accesses to the same shared variable in huge.

4.4 Logical Formalisms

Each logical formalism provided by the MOP framework is implemented as a program called a *logic plugin*. These logic plugins are available to JavaMOP, RV-Predict, and will be available to CMOP. BusMOP is able to use the ERE, FSM, LTL, and PTLTL plugins, only, as more powerful logics break the real time guarantees of BusMOP. An implementation of BusMOP using a gate would allow for context-free based properties. The syntax for each logical formalism can be found in [51].

Finite State Machines The finite state machine (FSM) plugin is one of the most important plugins for MOP. Not only is it a useful logical formalism in itself, but it is used as a backend for all logics reducible to finite automata. Currently, all logic plugins except for the context-free grammar (CFG) and past time linear temporal logic with calls and returns (PTCaRet) generate FSM output. This allows for a strong separation of concerns. For instance, minimization need occur only once, and it allows us to use one enable set generation algorithm for all of these plugins. Additionally, each instance of MOP need only know how to translate the pseudocode for FSMs, CFGs, and PTCaRet. Some MOP instances, such as BusMOP, may even opt to support only finite state monitors, in which case they only need to provide support for translating FSM pseudocode.[5]

An FSM property is a series of ⟨*Item*⟩s followed by ⟨*Alias*⟩s. An ⟨*Item*⟩ is essentially a state in the finite state machine, and the different transitions to take on a given input (⟨*Transition*⟩). The ⟨*Alias*⟩ allows for giving a name to a *set* of states. This is invaluable, because the ⟨*FSM State*⟩ non-terminal, which defines what categories may trigger handlers, both ⟨*Group Name*⟩s, which are the names associated to sets of states in ⟨*Alias*⟩s, and ⟨*State Name*⟩s may be associated with handlers. This allows one to write a property that triggers actions when any state in a given ⟨*Alias*⟩ is entered.

Figure 11 shows an example FSM property. In this example, three events: next, hasnext and dummy, and three states: start, safe and unsafe are defined. Two state aliases are declared: all_states represents all the states in the state machine and safe_states includes the start state and the safe state. The fail category is reported whenever an event occurs that is not specified for the current state. For example, the state machine will go into fail when the dummy event is seen in the unsafe state. The default transition in the start state covers any event not specified in the transition. Because of this, any state with a default transition cannot lead to a fail category for any input. Handlers may be associated with any state (e.g., start) or group name (e.g., all_states).

[5] Though note that in the case of BusMOP, finite state machines are not used for PTLTL in favor of using parallel assignments.

```
start [
  default start
  next -> unsafe
  hasnext -> safe
]
safe [
  next -> start
  hasnext -> safe
  dummy -> safe
]
unsafe [
  next -> unsafe
  hasnext -> safe
]
alias all_states = start, safe, unsafe
alias safe_states = start, safe
```

Figure 11. FSM Example

In the interest of keeping runtime monitoring as efficient as possible, we wish to use minimized finite state machines for monitors. Because of the ability to trigger handlers from ⟨*State Name*⟩s and ⟨*Group Name*⟩s, MOP FSM properties are *multicategory* finite state machines (finite state machines that recognize more than one language; essentially equivalent to Moore machines). This requires a small change to the normal Hopcroft FSM minimization algorithm [40].

The Hopcroft algorithm works by assuming the largest possible equivalence class of states, and then partitioning the equivalence classes into smaller classes if necessary. The way the algorithm determines that it is necessary to split is by considering two equivalence classes C_1 and C_2 and an input, e. For each state s in C_1, if s goes to a state in C_2 on e then it goes into class C_{11}, otherwise it goes to class C_{12}. Classes are continuously split by other classes until a fixed point is reached. When a fixed point is reached, each equivalence class becomes a state in the final machine.

The way our algorithm differs is in the initial partition. The normal algorithm partitions the states into two classes, those states that are final states and those which are not. We, however, have multiple categories. The particularly interesting feature, is that categories may overlap on states. If two categories C_1 and C_2 overlap, they must have three equivalence classes: those states in $C_1 - C_2$, those in $C_2 - C_1$, and those in $C_1 \cap C_2$. The naive algorithm would be to compute the intersections between all the categories, but that is quadratic in nature. A better algorithm, which we use, is to find the set of categories each state belongs to. This takes time linear in the number of states. Those states that have the same set of categories are placed in the same initial equivalence class. An in depth description of the algorithm can be found in [51].

Extended Regular Expressions Regular expressions can be easily understood by the average software engineer or programmer, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must not occur during an execution. Complementation gives one the power to express patterns on strings non-elementarily more compactly. Also, one important observation about the use of ERE in the context of runtime verification is that ERE patterns are often used to describe buggy patterns instead of desired properties.

Here is an example ERE property for the UnsafeMapIterator:

```
create_coll update_map* create_iter
    use_iter* update_map+ use_iter
```

Recall that in this property the sequence of actions of importance is the creation of an Iterator from a Collection that was created from a Map, which is updated between the creation of the Iterator and its use.

FSMs are generated from EREs using coinductive techniques [61]. Briefly, in our approach we use the concept of derivatives of a regular expression, which is based on the idea of event consumption, in the sense that an extended regular expression R and an event e produce another extended regular expression, denoted $R\{e\}$, with the property that for any trace w, trace $e\ w$ is in $\mathcal{L}(R)$ (i.e., the language denoted by R) iff w is in $\mathcal{L}(R\{e\})$. This technique produces a deterministic automaton, saving memory and time (in contrast to the more conventional Thompson approach [65], which operates by first producing a non-deterministic automaton, and then using a determinization algorithm to produce a deterministic automaton).

Context-Free Grammars Context-free grammars (CFG) are nearly as widely adopted by the average programmer as are regular expressions. Numerous context-free parser generators such as Bison [12] exist and are widely used. CFGs offer a level of expressibility greater than that of finite-monitor logics, and allow for the specification of properties that involve proper nesting and a notion of counting.

Below is an example CFG property.

```
S -> P endThread,
P -> P acquire P release | epsilon
```

In this example there are three events: acquire, release, and endThread, and one non-terminal, S. The fail category is reported whenever the program cannot have followed the proper lock nesting property, e.g., releasing a lock more times than it was acquired or not releasing it enough times before the end of the Thread. Additionally, we could handle match, which is reported whenever our locking discipline is faithfully completed.

The CFG plugin uses a generalized LR (GLR) parser. Specifically, when an ambiguity is encountered, instead of choosing one of the particular alternatives, all are tried in parallel. The parser accepts a string if any of its parallel parses does and rejects it if there is no possible parse in any of the alternate parses. Thus, unlike normal LR(1) parsers, the GLR algorithm is capable of recognizing all context-free languages. This parsing algorithm is both online and the overhead relative to a normal LR parser is proportional to the amount of ambiguity in the grammar. The LR parser tables are generated using Knuth's LR(1) parser table generation algorithm as presented in [2].

As with the other logic plugins we have the problem of not knowing when the last event of a trace slice will be seen. Thus, the plugin classifies traces into $\{\mathsf{match}, \mathsf{fail}, ?\}$. Originally, in [49], this was done by cloning the state of the monitor and seeing if the copy would accept on an end of trace event. After examining how the LR tables were constructed we noticed in [50] that if our parser is able to reduce on seeing an end of trace at all, then it must accept after some number of reductions. Thus, we just need to check if we can reduce assuming we are at the end of the trace instead of actually preforming the reductions. We refer to this concept as *guaranteed acceptance*. As a result, we no longer need to copy our parser's state and can just check whether the current state is the member of the set of states that would reduce at the end of the trace. More information on how to monitor CFGs can be found in [50].[6]

It should be noted that guaranteed acceptance is specific to monitoring and is *not* suitable for use in most parser generators since they do not need to only verify that a string is in the language but also to assemble an abstract syntax tree or produce some other side effects. Since these side effects are performed as part of the reductions omitting them is not, in general, possible. There are a number of open optimization opportunities, e.g., those suggested in [2, 4]. Additionally, there are GLR specific opportunities to share common segments of the copied stacks. Therefore, there is still room for generating better CFG monitors, though the current ones were satisfactory in our practical experiments.

Linear Temporal Logic with Past and Future Linear temporal logic (LTL) [58] is often used to specify properties in model checking. LTL formulae allow one to express concepts such as the occurrence of an event requiring that another event happen in the future. Note that runtime monitoring cannot guarantee the correctness of a safety or liveness property. Even though the properties might hold for a given execution of the system, they can only be proved to hold, in general, by exploring every possible state of the program. LTL specifications must be used with this in mind.

[6] Note that normal LR(1) and LALR(1) parsing are used in [50], while in the meanwhile, as presented here, we generalized the techniques in [50] to work with arbitrary context-free languages.

Below is an example LTL property, which states that all calls to next of an Iterator must be preceded by a call to hasnext, where hasnext returned true.

```
[](next => (*) hasnexttrue)
```

The algorithm for converting LTL with past and future to finite state machines suitable for monitoring uses the technique presented in [31] to produce a Buchii automata. Our implementation is the only implementation of this algorithm of which we are aware (the authors provide none), and corrects some bugs found in the paper. The Buchii automaton that results is then converted into a monitoring finite state machine using an implementation of the algorithm presented in [27]. The output is a non-minimal finite-state machine that is minimized by the FSM plugin.

Past Time Linear Temporal Logic Past time linear temporal logic is similar to LTL, except that all operators refer to only the past. Some safety properties are more easily expressed in terms of the past than the future, for example the property that a user authentication be required before accessing some resource is most naturally expressed as "access *implies* <*> authenticate", e.g., that an access requires an authentication at some point in the past. Monitors generated from PTLTL formulae also have the quality of validating or violating on every event because the past is already known. This contrasts with LTL monitors, which can also be in an intermediate, ?, state. Additionally, once an LTL monitor validates or violates, it is always violated or validated, whereas PTLTL is allowed to change on each event.

Below is an example PTLTL property. In this property the goal is to ensure that next is never called on an Iterator without first calling hasNext:

```
next => (*)hasNext
```

Note that, unlike the LTL example, there is no need for the [] always operator. Additionally, this property is capable of finding multiple violations during the run of the program, whereas the earlier LTL example can only find the first violation.

5 Research Plan

While the current results do much to prove the thesis, more need be accomplished. A string rewriting plugin will allow for formal specification of Turing complete properties, showing definitively that logics more powerful than finite state can be efficiently monitored. Making the interface for predicting generic properties in RV-Predict automatic will strengthen the result that generic safety properties can be predicted. A prototype for C Monitoring will show the feasibility of monitoring in another application domain—but one without garbage collection. Lastly, parallelizing the monitoring algorithm in JavaMOP will make JavaMOP even more efficient, hopefully to the point that we may be able to monitor hundreds of properties simultaneously with minimal overhead.

5.1 String Rewriting Plugin

String rewriting systems (SRS) use productions in a manner similar to context-free grammars, but rather than limiting the left hand side of productions to single non-terminals, any number of symbols may rewrite to any other symbols. String rewriting systems are well known to be Turing complete, and would give MOP its first Turing complete formal logic. Currently, JavaMOP has "raw" monitors, which use arbitrary Java code, but such is not a formal property, and is thus more prone to error.

The plugin will operate by maintaining a buffer of previously seen events. At such point that the buffer matches a left hand side of a production, a rewrite will be performed. Rewriting will happen as a fixed point until such point as no more productions may be applied to the buffer. The program may not continue until rewriting terminates if recovery actions are required by the system. More than likely, string rewriting systems will not be able to benefit from the enable and coenable optimizations of JavaMOP. Keeping the SRSs efficient will be a more difficult proposition because of this. Likely, a creative indexing scheme like that used in the Maude [25] interpreter will be necessary.

5.2 Predictive Analysis Interface

While RV-Predict is capable of using hand crafted general purpose monitors, it is necessary to provide a clean interface that provides general purpose monitors in order to make the system usable.

There are essentially two ways to monitor the causal model generated by RV-Predict. One method is to reverse the trace so that normal MOP monitoring semantics are maintained. Normal MOP semantics entail matching "good prefixes". What this means is that if, at a given point, the events seen so far match a property category, that category is reported. Unfortunately, reversing the trace is most likely prohibitively expensive, but we wish to make this an option within the tool. The other option is to reverse the monitor. This has the unfortunate effect of not having the ability to match good prefixes, as the trace is seen backward. However, this approach is much more efficient, and will also be supported by the tool. Our experiments thus far have focused on reversing the monitors and only checking for category matches once the entire trace has been processed.

5.3 C/C++ Monitoring Prototype

We have intended to extend MOP to the C/C++ language for a while, and doing so further strengthens our goal of efficient parametric monitoring in the application domain. While there have been some attempts to monitor C in the past [16, 33], most monitoring systems have focused on Java. These two previous attempts also use hardwired logical formalisms.

ApectiveC++ [64] will be used by the system in the same way as AspectJ is used by JavaMOP. To cut down on implementation time a full C/C++ parser will not be used, rather handler code will be pasted "as is", with AspectiveC++ left to find any syntax errors. A fully functional system complete with parsing could not be completed in the alloted time frame.

New algorithms will need be designed for use with CMOP because we will not be able to rely on Java's garbage collection to help us remove many of our dead monitor instances. These algorithms will be part of the prototype.

5.4 Separate Thread Monitoring

We are all well aware that the future—the present really—of computer hardware is more and more cores on a die. We are, thus, left with an ever increasing amount of hardware threads. Currently, monitoring slows down single threaded performance. We wish to leverage the newly available hardware threads by placing monitoring code outside of the main program thread, potentially up to one thread per property.

There are several hurdles to the approach, however. If all a user worries about is finding errors via messages, then all we need do is monitor in a separate thread, no caveats necessary. However, many of our monitors need to effect recovery action within the program. Obviously, the program cannot be allowed to continue in places where an error can occur if recovery is to be effective. We use standard terminology, and refer to this as synchronization between the monitoring code and thread. As a naive option we can simply synchronize on every event that may cause a handler to run (i.e., an event that *may* lead to a state that triggers a handler), if said handler is marked as requiring synchronization by the user. A more runtime efficient idea would be using something like the Nop-Shadows optimization presented in [15] in order to synchronize in as few places as can be statically determined. What a small modification of Nop-Shadows would tell us is the exact program points where we need to synchronize for an event e that may lead to a handler, rather than synchronizing at *all* program points where e occurs.

Separate thread monitoring should be extendible to CMOP, however, an entire framework similar to Clara will need be developed for AspectiveC++.

5.5 Summary of Research Plan

We summarize the research as follows. The total new research is expected to be completed within roughly five months, in time for May 2012.

- String Rewriting System Plugin - 2 months
  - Initial Implementation - $\frac{1}{2}$ month
  - Optimization - $1\frac{1}{2}$ months
- Predictive Analysis Interface - 1 month
  - Trace Reversal and Monitor Reversal - $\frac{1}{2}$ month
  - Providing RV-Predict GUI support - $\frac{1}{2}$ month
- CMOP - 1 month
  - Initial Implementation - $\frac{1}{2}$ month
  - Garbage Collection and Optimization - $\frac{1}{2}$ month
- Separate Thread Monitoring - 1 month
  - Initial Implementation - $\frac{1}{2}$ month
  - Interfacing with NOP-Shadows and perhaps other Optimizations - $\frac{1}{2}$ month

6 Related Work

We next discuss relationships between the MOP framework and other related paradigms, including AOP, design by contract, runtime verification, and other trace monitoring approaches. Broadly speaking, all the approaches discussed below are instances of runtime monitoring. Interestingly, even though most of the systems mentioned below target the same programming languages, no two of them share the exact same logical formalism for expressing properties. This observation strengthens our belief that probably there is *no silver bullet logic* (or *super logic*) for all purposes. A major objective in the design of the MOP framework was to avoid hardwiring particular logical formalisms into the system.

6.1 Aspect Oriented Programming (AOP) Languages

Since its proposal in [43], AOP has been increasingly adopted and many tools have been developed to support AOP in different programming languages, e.g., AspectJ and JBoss [41] for Java, and AspectC++ [5] for C++. Built on these general AOP languages, numerous extensions have been proposed to provide domain-specific features for AOP. Among these extensions, Tracematches [3] and J-LO [14] support history(trace)-based aspects for Java.

Tracematches enables the programmer to trigger the execution of certain code by specifying a parametric regular pattern of events in a computation trace, where the events are defined over entry/exit of AspectJ pointcuts. When the pattern is matched during the execution, the associated code will be executed. In this sense, Tracematches supports trace-based pointcuts for AspectJ. J-LO is a tool for runtime-checking temporal assertions. These temporal assertions are specified using parametric linear temporal logic (LTL) and the syntax adopted in J-LO is similar to Tracematches' except that the properties are specified in a different formalism. J-LO also uses the same parametricity semantics as Tracematches. J-LO mainly focuses on checking at runtime properties rather than providing programming support. In J-LO, the temporal assertions are inserted into Java files as annotations that are then compiled into runtime checks. Both Tracematches and J-LO support parametric events, i.e., free variables can be used in the specified properties and will be bound to specific values at runtime for matching events.

The MOP framework has logic plugins, which encapsulate different logical formalisms and allow it to capture the capabilities of Tracematches and J-LO. JavaMOP is the instantiation of the MOP framework for Java programs.

JavaMOP allows for two different modes of matching traces, referred to as total trace matching and suffix trace matching. Total is the default mode of JavaMOP, while suffix mode is used by prefixing a JavaMOP property with the suffix modifier (see [51]).

With total matching, for example, with the pattern a^*b, a sequence of events abb will trigger the validation handler of the generated MOP monitor only at the first b event and then the violation handler (if any) at the second b.

With suffix matching, however, the pattern will be matched twice, once for each b event: the first matches either the whole trace a b or the partial trace consisting of just the first b with zero occurrences of a, while the second matches

the subsequent partial trace b (the second b in the trace) with zero occurrences of a; thus, the related advice will be executed twice.

With suffix matching one can count matches of a pattern open close without a need to reset the monitor after each match, as would be required with total match monitoring. On the other hand, total trace matching is more suitable for runtime verification of formal properties, because it is the only semantics that makes sense for some logical formalisms, such as LTL, and thus many users expect this behavior for pattern languages like regular expressions and context-free grammars, as well.

J-LO can be captured by the JavaMOP with total matching because LTL is supported by the MOP framework. MOP supports regular expressions as part of its extended regular expression (ERE) logic plugin, and Tracematches may be captured by JavaMOP by using these ERE patterns with suffix matching.

6.2 Runtime Verification

In runtime verification, monitors are automatically synthesized from formal specifications, and can be deployed *offline* for debugging, or *online* for dynamically checking properties during execution. MaC [44], PathExplorer (PaX) [36], Eagle [9], and RuleR [10] are runtime verification frameworks for logic based monitoring, within which specific tools for Java – Java-MaC, Java PathExplorer, and Hawk [26], respectively – are implemented. All these runtime verification systems work in outline monitoring mode and have hardwired specification languages: MaC uses a specialized language based on interval temporal logic, JPaX supports just LTL, and Eagle adopts a fixed-point logic. Java-MaC and Java PathExplorer integrate monitors via Java bytecode instrumentation, making them difficult to port to other languages. Our MOP approach supports inline, outline, and offline monitoring; allows one to define new formalisms to extend the MOP framework; and is adaptable to new languages (we discuss two such instances in this paper).

Temporal Rover [28] is a commercial runtime verification tool based on future time metric temporal logic. It allows programmers to insert formal specifications in programs via annotations, from which monitors are generated. An Automatic Test Generation (ATG) component is also provided to generate test sequences from logic specifications. Temporal Rover and its successor, DB Rover, support both inline and offline monitoring. However, they also have their specification formalisms hardwired and are tightly bound to Java. MOP currently has no metric temporal logic plugin.

6.3 Design by Contract

Design by Contract (DBC) [53] is a technique allowing one to add semantic specifications to a program in the form of assertions and invariants, which are then compiled into runtime checks. It was first introduced as a built-in feature of the Eiffel language [29]. Some DBC extensions have also been proposed for a number of other languages. Jass [11] and jContractor [1] are two Java-based approaches.

Jass is a precompiler which turns the assertion comments into Java code. Besides the standard DBC features such as pre-/post- conditions and class invariants, it also provides refinement checks. The design of trace assertions in Jass is mainly influenced by CSP [39], and the syntax is more like a programming language. jContractor is implemented as a Java library which allows programmers to associate contracts with any Java class or interface. Contract methods can be included directly within the Java class or written as a separate contract class. Before loading each class, jContractor detects the presence of contract code patterns in the Java class bytecode and performs on-the-fly bytecode instrumentation to enable checking of contracts during the program's execution. jContractor also provides a support library for writing expressions using predicate logic quantifiers and operators such as *Forall*, *Exists*, *suchThat*, and *implies*. Using jContractor, the contracts can be directly inserted into the Java bytecode even without the source code.

Java modeling language (JML) [46] is a behavioral interface specification language for Java. It provides a more comprehensive modeling language than DBC extensions. Not all features of JML can be checked at runtime; its runtime checker supports a DBC-like subset of JML. Spec# [7] is a DBC-like extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions and also provides method contracts in the form of pre- and post-conditions as well as object invariants. Using the Spec# compiler, one can statically enforce non-null types, emit run-time checks for method contracts and invariants, and record the contracts as metadata for consumption by downstream tools.

We believe that the logics of assertions/invariants used in DBC approaches fall under the uniform format of our logic engines, so that an MOP environment following our principles would naturally support monitoring DBC specifications as a special methodological case. In addition, the MOP framework also supports outline monitoring, which we find

important in assuring software reliability (e.g., monitoring for and detecting and fixing deadlocks) but which is not provided by any of the current DBC approaches that we are aware of.

6.4 Other Monitoring Approaches

Program Query Language (PQL) allows programmers to express design rules that deal with sequences of events associated with a set of related objects [48]. Both static and dynamic tools have been implemented to find solutions to PQL queries. The static analysis conservatively looks for potential matches for queries and is useful to reduce the number of dynamic checks. The dynamic analyzer checks the runtime behavior and can perform user-defined actions when matches are found. PQL has a "hardwired" specification language based on context-free grammars (CFG) and supports only inline monitoring. CFGs can potentially express more complex languages than regular expressions, so in principle PQL can express more complex safety policies than Tracematches. The MOP CFG plugin described in Section 4.4 allows the MOP framework to specify most of the properties that may be specified in PQL.

Program Trace Query Language (PTQL) [32] is a language based on SQL-like relational queries over program traces. The current PTQL compiler, Partiqle, instruments Java programs to execute the relational queries on the fly. PTQL events are timestamped and the timestamps can be explicitly used in queries. PTQL queries can be arbitrarily complex and, as shown in [32], PTQL's runtime overhead seems acceptable in many cases but we were unable to obtain a working package of PTQL and compare it in our experiments with JavaMOP because of license issues. PTQL properties are globally scoped and their running mode is inline. PTQL provides no support for recovery, its main use being to detect errors.

6.5 Concurrency Error Detection

There are several other approaches also aiming at detecting potential concurrency errors by examining particular execution traces. Some of these approaches aim at verifying general purpose properties [60, 62], including temporal ones, and are inspired from debugging distributed systems based on Lamport's *happens-before* causality [45]. Other approaches work with particular properties, such as data-races and/or atomicity. [59] introduces a first lock-set based algorithm to detect data-races dynamically, followed by many variants aiming at improving its accuracy. For example, an ownership model was used in [66] to achieve a more precise race detection at the object level. [54] combines the lock-set and the happen-before techniques. The lock-set technique has also been used to detect atomicity violations at runtime, e.g., the reduction based algorithms in [30] and [67]. [67] also proposes a block-based algorithm for dynamic checking of atomicity built on a simplified happen-before relation, as well as a graph-based algorithm to improve the efficiency and precision of runtime atomicity analysis.

Previous efforts tend to focus on either soundness or coverage: those based on happens-before try to be sound, but have limited coverage over interleavings, thus missing errors; lock-set based approaches have better coverage but suffer from false alarms. RV-Predict aims at improving coverage without giving up soundness or genericity of properties. It combines *sliced causality* [21], a happen-before causality drastically but soundly sliced by removing irrelevant causalities using semantic information about the program obtained with an apriori static analysis, with *lock-atomicity*.

6.6 Hardware-based Monitoring

The PSL to Verilog compiler, P2V [47], is the sole attempt to perform runtime monitoring of *formal properties* in hardware, other than BusMOP, of which we are aware. P2V is similar to BusMOP in that monitors are implemented in hardware rather than software, and that both approaches thus have no runtime overhead on the CPU. P2V, however, is more like the above approaches in that it is designed for monitoring actual programs rather than peripheral devices. Also it requires a dynamically extensible soft-core processor implemented on an FPGA, while our approach can potentially be applied to any COTS communication architecture. Further, P2V uses hardwired logic (PSL) while BusMOP allows multiple formalisms.

7 Conclusion

The proposed research, as well as the already complete research, shows that runtime monitoring techniques can be applied to a large number of domains, and that the monitored logics need not be only finite-state. Java and C/C++

monitoring and the optimizations thereof show that monitoring can be efficient within the application domain. BusMOP shows that monitoring can be used to ensure the proper operation of peripherals in a complex system. The use of monitoring techniques in predictive analysis shows that the monitored execution trace need not result from an actually observed run of a program. The context-free and string rewriting algorithms emphatically show that monitoring logics need not be finite-state in order to be efficient.

References

1. P. Abercrombie and M. Karaorman. jContractor: Bytecode instrumentation techniques for implementing DBC in Java. In *Runtime Verification (RV'02)*, volume 70 of *ENTCS*. Elsevier, 2002.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986. pages 215–246.
3. C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 345–364. ACM, 2005.
4. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004.
5. AspectC++. `http://www.aspectc.org/`.
6. P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 589–608. ACM, 2007.
7. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
8. H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors. *Runtime Verification (RV'05)*, volume 144 of *ENTCS*. Elsevier, 2005.
9. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
10. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RULER. *J. Logic Computation*, November 2008.
11. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass-Java with Assertions. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*, pages 103–117. Elsevier, 2001.
12. Bison. `http://www.gnu.org/software/bison/`.
13. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, pages 169–190. ACM, 2006.
14. E. Bodden. J-LO, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.
15. E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *International Conference on Software Engineering (ICSE'10)*, pages 5–14, 2010.
16. S. Chaudhuri and R. Alur. Instrumenting c programs with nested word monitors. In *Model Checking Software (SPIN'07)*, volume 4595 of *LNCS*, pages 279–283. Springer, 2007.
17. S. Chaudhuri and R. Alur. Instumenting C programs with nested word monitors. In *Model Checking Software (SPIN'07)*, volume 4595 of *LNCS*, pages 279–283. Springer, 2007.
18. F. Chen, P. O. Meredith, D. Jin, and G. Roşu. Efficient formalism-independent monitoring of parametric properties. In *Automated Software Engineering (ASE'09)*, pages 383–394. IEEE, 2009.
19. F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 108–127. Elsevier, 2003.
20. F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 569–588. ACM, 2007.
21. F. Chen and G. Roşu. Parametric and sliced causality. In *Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 240–253. Springer, 2007.
22. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261. Springer, 2009.
23. F. Chen and G. Roşu. Parametric and termination-sensitive control dependence - extended abstract. In *Static Analysis Symposium (SAS'06)*, 2006.
24. F. Chen, T. F. Şerbănuţă, and G. Roşu. jPredictor: a predictive runtime analysis tool for Java. In *International Conference on Software Engineering (ICSE'08)*, pages 221–230. ACM, 2008.

25. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Springer-Verlag New York, Inc., 2007.
26. M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.
27. M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. In *Proceedings of 17th International Conference on Computer-aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 364 – 378. Springer, 2005.
28. D. Drusinsky. The Temporal Rover and the ATG Rover. In *Model Checking and Software Verification (SPIN'00)*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
29. Eiffel Language. `http://www.eiffel.com/`.
30. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Principles of Programming Languages (POPL'04)*, 2004.
31. P. Gastin and D. Oddoux. Ltl with past and two-way very-weak alternating automata. In *Mathematical Foundations of Computer Science (MFCS'03)*, pages 439–448, 2003.
32. S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 385–402. ACM, 2005.
33. K. Havelund. Runtime verification of c programs. In *TestCom/FATES*, pages 7–22, 2008.
34. K. Havelund, M. Nunez, G. Roşu, and B. Wolff, editors. *Formal Approaches to Testing and Runtime Verification (FATES/RV'06)*, volume 4264 of *LNCS*. Springer, 2006.
35. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*, pages 97–114. Elsevier, 2001.
36. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.
37. K. Havelund and G. Roşu, editors. *Runtime Verification (RV'02)*, volume 70 of *ENTCS*. Elsevier, 2002.
38. K. Havelund and G. Roşu, editors. *Runtime Verification (RV'04)*, volume 113 of *ENTCS*. Elsevier, 2004.
39. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall Intl., New York, 1985.
40. J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford University, 1971.
41. JBoss. `http://www.jboss.org`.
42. D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, pages 415–424. ACM, 2011.
43. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.
44. M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Europoean Conference on Real-Time Systems (ECRTS'99)*, 1999.
45. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of ACM*, 21(7):558–565, 1978.
46. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, pages 105–106. ACM, 2000.
47. H. Lu and A. Forin. The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Technical Report MSR-TR-2007–99, Microsoft Research, 2007.
48. M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 365–383. ACM, 2005.
49. P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE '08)*, pages 148–157. IEEE, 2008.
50. P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *J. Automated Software Engineering*, pages 149–180, 2010.
51. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011. To appear; http://dx.doi.org/10.1007/s10009-011-0198-6.
52. P. O. Meredith and G. Rosu. Runtime verification with the rv system. In *Runtime Verification (RV'10)*, pages 136–152, 2010.
53. B. Meyer. *Object-Oriented Software Construction,* 2^{nd} *edition*. Prentice Hall, New Jersey, 2000.
54. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, 2003.
55. R. Pellizzoni, B. D. Buy, M. Caccamo, and L. Sha. Coscheduling of real-time tasks and PCI bus transactions. Technical report, University of Illinois at Urbana-Champaign, 2008. Available at `http://netfiles.uiuc.edu/rpelliz2/www/techreps/`.
56. R. Pellizzoni, P. O. Meredith, M. Caccamo, and G. Roşu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time System Symposium (RTSS'08)*, pages 481–491. IEEE, 2008.
57. R. Pellizzoni, P. O. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed-criticality in soc-based real-time embedded systems. In *Embedded Software (Emsoft'09)*, pages 235–244, 2009.

58. A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE, 1977.
59. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transaction of Computer System*, 15(4):391–411, 1997.
60. A. Sen and V. K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *International Conference on Principles of Distributed Systems (OPODIS'03)*, 2003.
61. K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 162–181. Elsevier, 2003.
62. K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'03)*, 2003.
63. O. Sokolsky and M. Viswanathan, editors. *Runtime Verification (RV'03)*, volume 89 of *ENTCS*. Elsevier, 2003.
64. O. Spinczyk, D. Lohmann, and M. Urban. Advances in aop with aspectc++. In *New Trends in Software Methodologies, Tools and Techniques (SoMeT'05)*, pages 33–53, 2005.
65. K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.
66. C. von Praun and T. R. Gross. Object race detection. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, 2001.
67. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, 2006.
68. Xilinx, Inc. *Virtex-4 ML455 PCI/PCI-X Development Kit User Guide*. `http://www.xilinx.com/support/documentation/boards_and_kits/ug084.pdf`.